Hashing and Message Authentication

Daniel Zappala

CS 465 · Introduction to Security and Privacy · Fall 2026

BYU



01

Hashing

Fixed-length fingerprints of any data

01

Hashing

A hash maps input of any length to a fixed-length digest

Infeasible to reverse



SHA-256

hash function

```
e3b0c44298fc1c14
9afbf4c8996fb924
27ae41e4649b934c
a495991b7852b855
```

Any input

A password, a file or a 4 GB disk image

Fixed output

SHA-256 always returns 256 bits (64 hex characters)

Same input, same digest

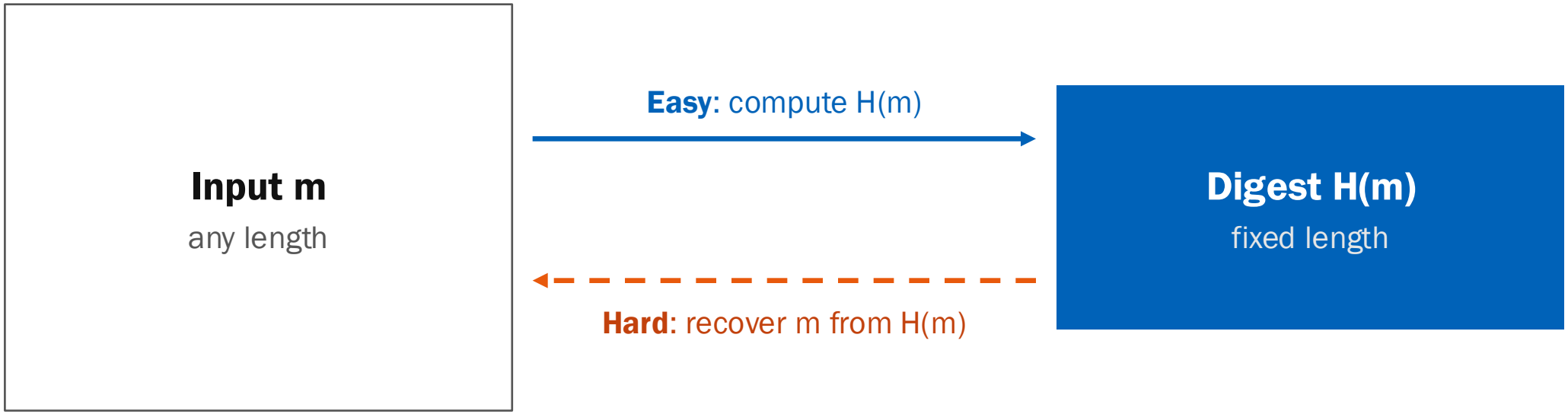
Anyone can recompute it; no key is involved

“

Probably the single most useful tool in a cryptographer’s toolbox

Bruce Schneier, on one-way hash functions

A hash is easy to compute and hard to invert



Computing a digest takes microseconds; the only known way back is guessing inputs

Adapted from P. C. van Oorschot, Computer Security and the Internet, Fig. 2.10

01

Hashing

Uses of hash functions



Password storage

Servers store a hash, never the password itself



Download checks

Compare a file's SHA-256 against the digest published by its maker



Digital signatures

Sign the short digest instead of the whole document



Deduplication

Cloud storage spots identical files by comparing their hashes



Version control

Git names every commit and file by its hash



Blockchains

Each block includes the hash of the block before it

01

Hashing

Changing one input bit flips about half the output bits

SHA-256 digests of two inputs one character apart

"cat"	`77af778b51abd4a3c51c5ddd97204a9c3ae614ebccb75a606c3b6865aed6744e`
"bat"	`ca5bcec12f716f44d9745d349cc80422f0d14cbab09329caf533bef7c2d952eb`

129 of 256

output bits differ (50%)

Avalanche effect: similar inputs give unrelated digests, so a digest reveals nothing about which input is close to the original

01

Hashing

Collision resistance is the strongest guarantee, yet the cheapest to attack

Each step implies the one below it: breaking step 1 breaks step 2, and breaking step 2 breaks step 3

Also called: One-way

1

Preimage resistance

Given y, find any m where H(m) = y

Attacker is given y
Chooses m

SHA-256 attack cost
≈ 2^{256} work

Also called: Weak collision resistance

2

Second preimage resistance

Given m, find a different m′ with H(m′) = H(m)

Attacker is given m
Chooses m′

SHA-256 attack cost
≈ 2^{256} work

Also called: Strong collision resistance

3

Collision resistance

Find m ≠ m′ where H(m) = H(m′)

Attacker is given nothing
Chooses both m and m′

SHA-256 attack cost, birthday bound
≈ 2^{128} work

01

Hashing

Each use of a hash depends on a different property

Use	Property it needs	Why
Password storage Store H(password), compare at login	**Preimage resistance**	A stolen digest file must not reveal passwords
File integrity (Tripwire) Record digests of system files, recheck later	**Second preimage resistance**	An intruder must not swap in a file with the same digest; collision resistance if the attacker picks the original file
Digital signatures Sign H(m) instead of m	**Collision resistance**	A signer, or someone who wrote m, must not find two messages with one digest

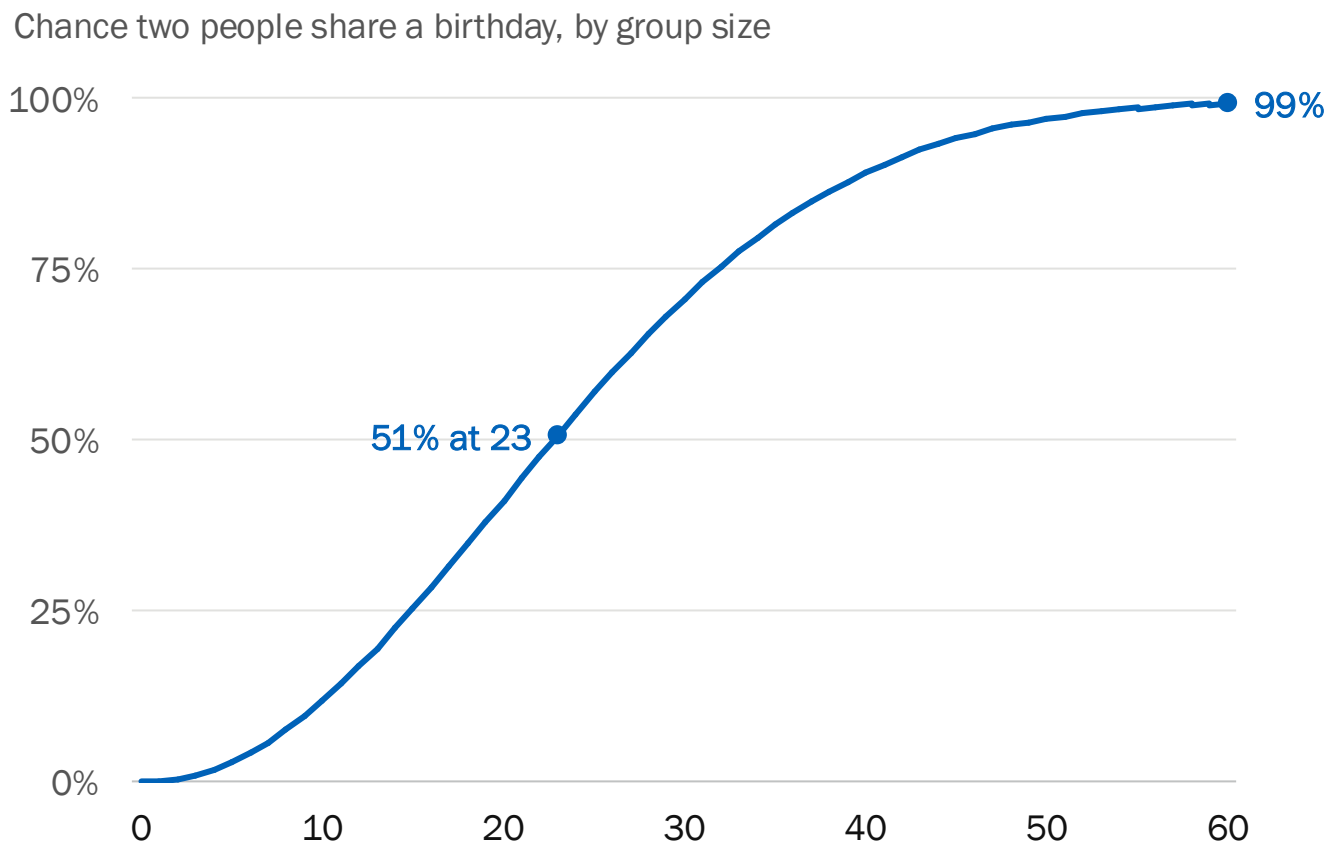


How many people need to be in a room for even odds that two share a birthday?

01

Hashing

The birthday paradox makes collisions far easier to find



23 people

are enough for a 50% chance of a shared birthday, out of 365 days

For an n-bit hash, a collision takes about $2^{n/2}$ tries, not 2^{n}

SHA-256: about 2^{128} work, still out of reach

MD5: 128 bits, so 2^{64} at best, and real attacks are far faster

01

Hashing

MD5 and SHA-1 were both broken in the same way

	WEAKNESS FOUND	FIRST FULL COLLISION	CHOSEN-PREFIX COLLISION	REAL-WORLD IMPACT
MD5 published 1992	**1996** **Compression flaw** Dobbertin finds collisions in the MD5 compression function	**2004** **First MD5 collisions** Wang and colleagues find full collisions in hours	**2008** **Rogue CA certificate** A chosen-prefix collision and 200 PS3s forge a CA that browsers trust	**2012** **Flame malware** Forges a Microsoft code-signing certificate to spread through Windows Update
SHA-1 published 1995	**2005** **First SHA-1 attack** Wang, Yin and Yu find collisions in about 2^{69} work, far below 2^{80}	**2017** **Colliding PDFs** Google and CWI publish SHAttered; browsers stop accepting SHA-1 certificates	**2019** **Chosen-prefix attack** Leurent and Peyrin: chosen-prefix collisions for about $100K	**2020** **Windows drops SHA-1** Microsoft retires SHA-1 signing for update delivery



A server stores SHA-256(password). An attacker steals the database. What next?

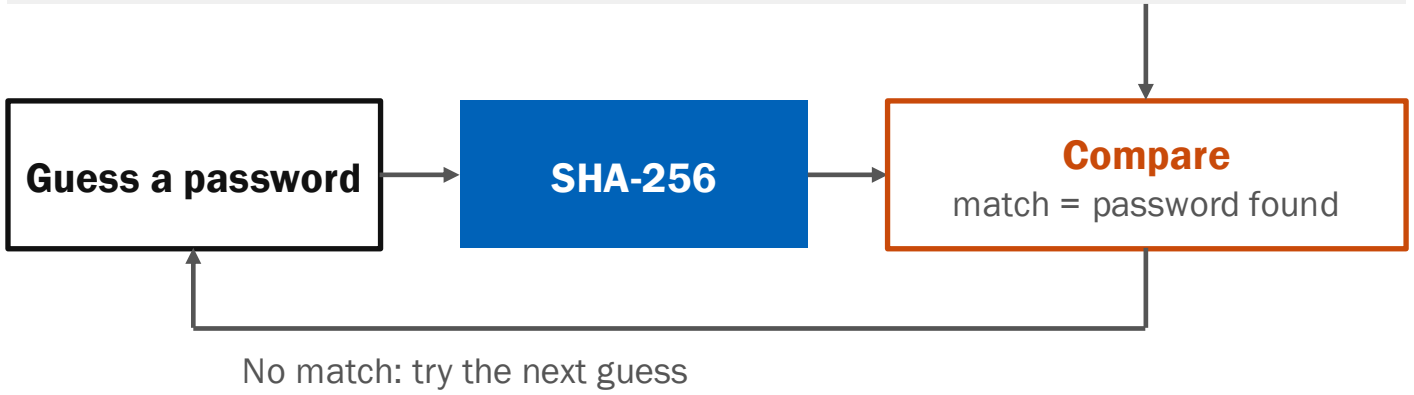
01

Hashing

A fast hash lets an attacker test billions of password guesses per second

Offline guessing, after the database is stolen

Stolen row: `alice  5e884898da280471…`



Guesses come from **leaked password lists, dictionaries and common patterns**, so most real passwords fall long before an exhaustive search would finish

≈ 20 seconds

to try every 8-letter lowercase password at 10 billion SHA-256 guesses per second

No salt? Users with the same password share a hash, and precomputed tables crack them all at once

Fix: a unique salt per user and a slow, memory-hard hash such as Argon2id

$26^8 \approx 2.1 \times 10^{11}$ candidates. The guess rate assumes one modern GPU and varies with hardware

01

Hashing

Use SHA-2, SHA-3 or BLAKE for data, and a slow hash for passwords

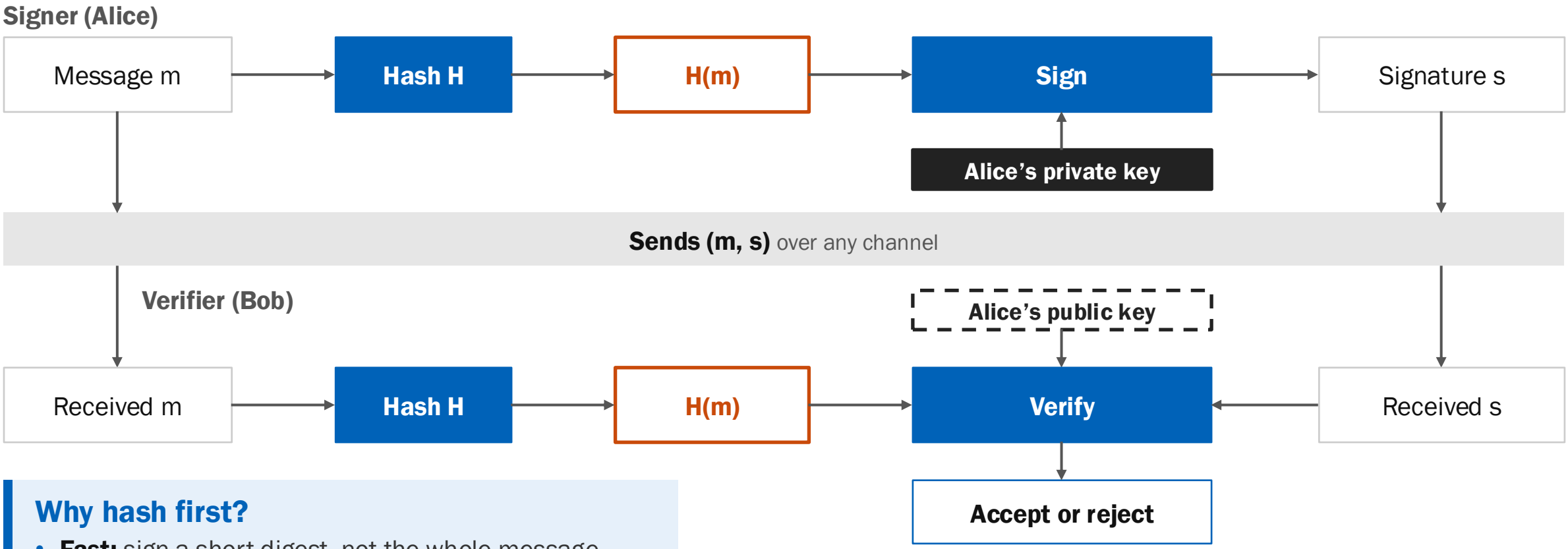
Hash function	Digest (bits)	Block (bits)	Status
For data			
SHA-224 / SHA-256	224 / 256	512	SHA-2 family, in wide use
SHA-384 / SHA-512	384 / 512	1024	SHA-2 family, in wide use
SHA3-256 / SHA3-512	256 / 512	1088 / 576	SHA-3 (Keccak), 2015 standard
SHAKE128 / SHAKE256	Any length	1344 / 1088	SHA-3 family; used in ML-KEM, ML-DSA
BLAKE2b / BLAKE2s	≤512 / ≤256	1024 / 512	Secure; in WireGuard and Argon2
BLAKE3	256, extendable	512	Secure, 2020; OpenZFS, ccache
For passwords			
Argon2	Any length	—	Passwords: first choice (**Argon2id**)
scrypt	Any length	—	Passwords: fallback if no Argon2id
bcrypt	184	—	Passwords: legacy; 72-byte limit
PBKDF2	Any length	—	Passwords: when FIPS is required

Slow on purpose: password hashes are deliberately slow and memory-hard, so every guess costs an attacker time and RAM

01

Hashing

Digital signatures are computed on the hash, not the message



Why hash first?

- **Fast:** sign a short digest, not the whole message
- **Safe:** collision resistance makes H(m) stand for m

Adapted from P. C. van Oorschot, Computer Security and the Internet, Fig. 2.11

02

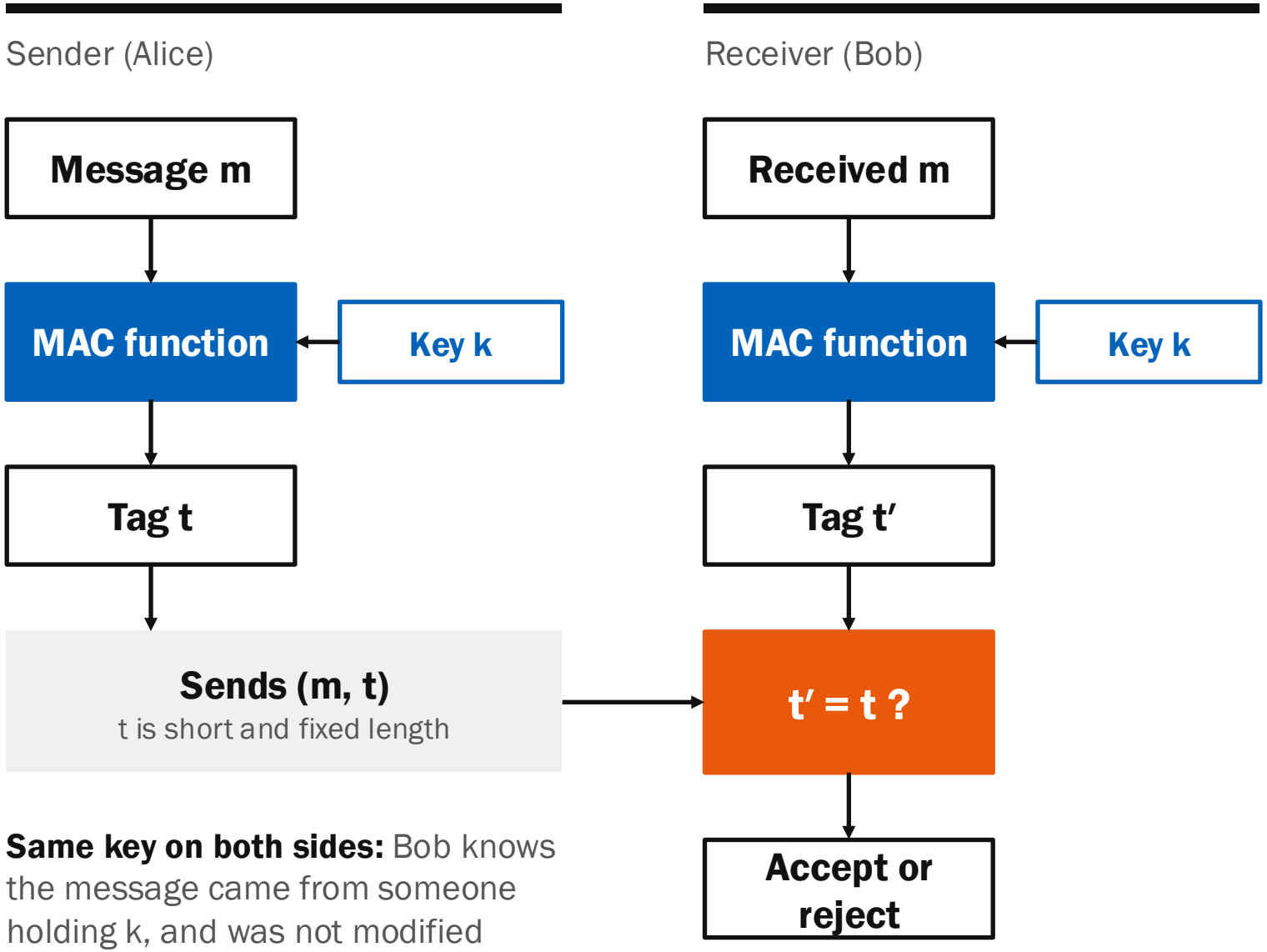
Message authentication

Integrity and origin from a shared key

02

Message authentication

A MAC proves integrity and origin to whoever holds the key



Same key on both sides: Bob knows the message came from someone holding k, and was not modified

Adapted from P. C. van Oorschot, Computer Security and the Internet, Fig. 2.12

?

Can Bob prove to a judge that a MAC tag came from Alice?

02

Message authentication

A MAC can’t prove to a third party who sent a message

	MAC	Digital signature
Keys	One shared secret key	Private key signs, public key verifies
Integrity	Yes	Yes
Origin, to the receiver	Yes: someone holding k	Yes: holder of the private key
Proof to a third party	**No: either party could have made the tag**	**Yes: only the signer could have signed**
Speed	Fast (symmetric)	Slower (public key)

MACs and digital signatures don’t stop replays

An attacker can resend a valid (m, t) later. Add a sequence number, timestamp or nonce under the MAC so the receiver can tell the message is fresh

?

Can a MAC be just a hash of the key and the message?

tag = H(k ‖ m) with SHA-256

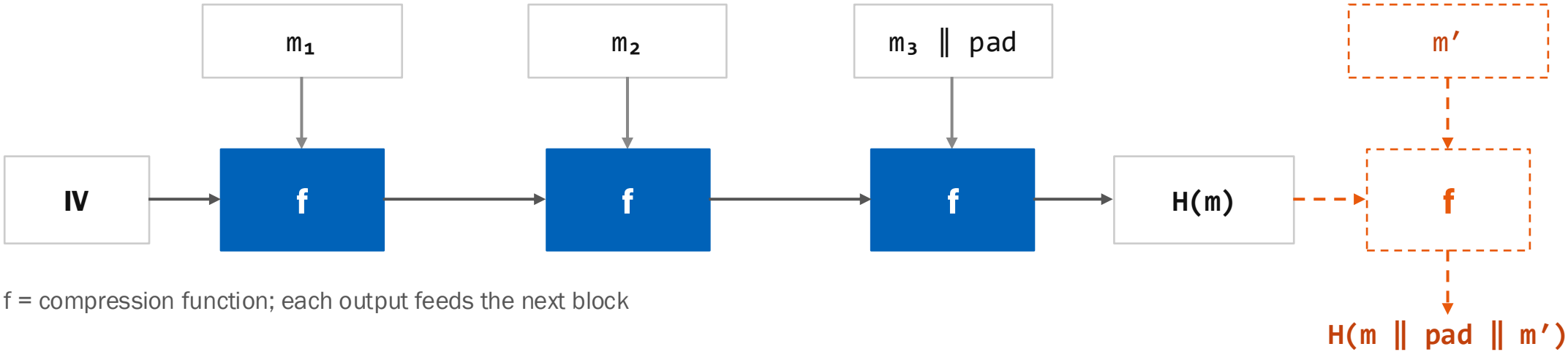
What could an attacker do with one valid tag?

02

Message authentication

The obvious MAC, H(k ‖ m), fails because SHA-256 chains blocks (Merkle–Damgård)

Goal: build a MAC from a hash **First try:** `tag = H(k ‖ m)`



f = compression function; each output feeds the next block

1 H(m) is the whole chain state

The digest is simply what the last f outputs. Nothing about the chain is hidden or sealed at the end

2 Anyone can resume the chain

Use H(m) as the chaining value and run f on a new block m′. No knowledge of m is needed, only its length to rebuild the padding

3 The result is a valid hash

The output equals H(m ‖ pad ‖ m′). So a tag computed as H(k ‖ m) lets an attacker forge a tag for a longer message

Based on M. Rosulek, The Joy of Cryptography, ch. 11

02

Message authentication

Putting a key into a hash is harder than it looks

Secret prefix

$H(k \| m)$

Length extension: given $H(k \| m)$, anyone can compute $H(k \| m \| \text{pad} \| m')$ in MD5, SHA-1 and SHA-256

Breaks: unforgeability

Tags forged without the key, even though SHA-256 itself is not broken

Secret suffix

$H(m \| k)$

A collision on m gives a forgery: if $H(m) = H(m')$ then the internal state matches and the tag for m also works for m'

Breaks: unforgeability

Any collision in H gives a forgery; a birthday search finds one in about $2^{n/2}$ work

Envelope

$H(k \| m \| k)$

Better, but its security was never cleanly proven, and published attacks recover key material faster than brute force

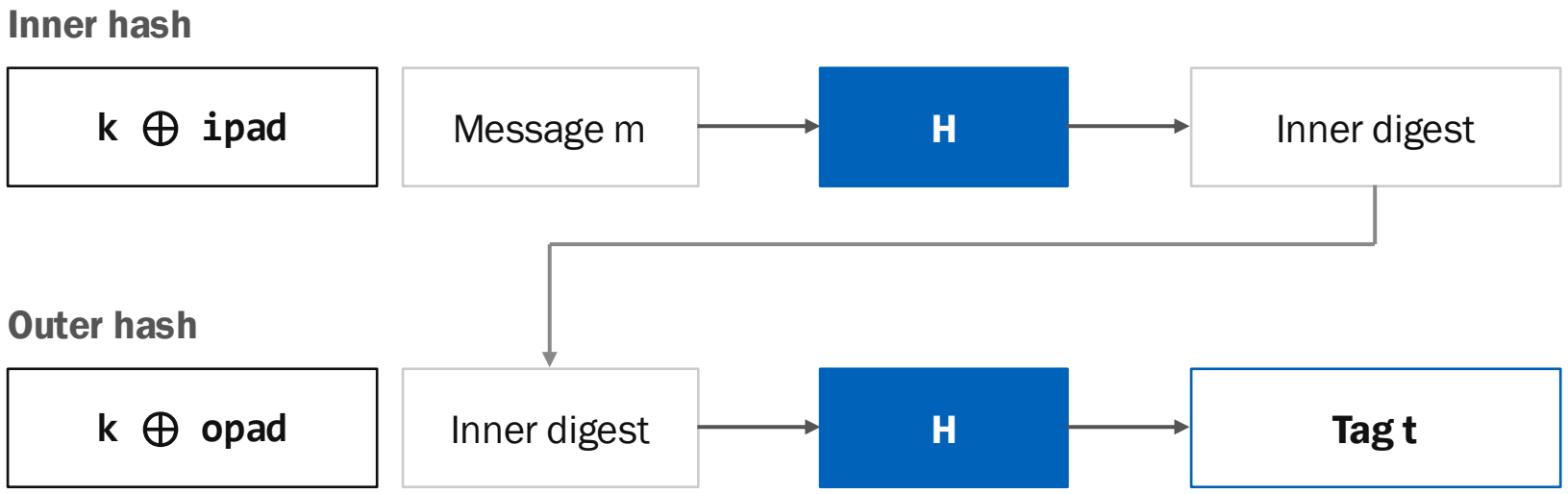
Breaks: key secrecy

Attacks recover the key itself (Preneel and van Oorschot)

HMAC fixes this with two nested hashes and a proof that rests on the hash's own properties

HMAC hashes twice, with two padded versions of the key

$$\mathrm{HMAC}(k, m) = H((k \oplus \mathrm{opad}) \parallel H((k \oplus \mathrm{ipad}) \parallel m))$$



ipad = 0x36 repeated, opad = 0x5C repeated, each one block long

The outer hash hides the inner state, which blocks length extension

Collisions don't break it: HMAC is secure if the hash's compression function looks random under a secret key, so HMAC-SHA1 survived SHA-1's collisions

In practice: HMAC-SHA256 in TLS 1.2, JWTs (HS256) and AWS request signing

02

Message authentication

Use HMAC for a MAC alone, AEAD when you also encrypt

Integrity against an attacker

HMAC-SHA256

Message stays readable; you only need to detect tampering

- API request signing
- Cookies and tokens
- Key derivation (HKDF)

Confidentiality + integrity

AEAD: AES-GCM

One call encrypts and authenticates; also covers unencrypted header data

- TLS 1.3 records
- Disk and file encryption
- ChaCha20-Poly1305 on devices without AES hardware

Class exercise: hashing with Rust Crypto

- **Crates:** Rust Crypto, SHA2, SHA3
- **Code examples:** hash-example in the Rust Cryptography repo